Scheduling of CAL actor networks based on dynamic code analysis

Jani Boutellier, Olli Silven, Mickaël Raulet

SCHEDULING OF CAL ACTOR NETWORKS BASED ON DYNAMIC CODE ANALYSIS

Jani Boutellier, Olli Silvén

University of Oulu
Department of Computer Science and Engineering
FI-90014 University of Oulu

Mickaël Raulet

INSA Rennes
IETR
FR-35708 Rennes Cedex 7

ABSTRACT

CAL is a dataflow oriented language for writing high-level specifications of signal processing applications. The language has recently been standardized and selected for the new MPEG Reconfigurable Video Coding standard.

Application specifications written in CAL can be transformed into executable implementations through development tools. Unfortunately, the present tools provide no way to schedule the CAL entities efficiently at run-time.

This paper proposes an automated approach to analyze specifications written in CAL, and produce run-time schedules that perform on average 1.5x faster than implementations relying on default scheduling. The approach is based on quasi-static scheduling, which reduces conditional execution in the run-time system.

***Index Terms*—** Scheduling, signal processing, data flow analysis

1. INTRODUCTION

Reconfigurable Video Coding (RVC) is a relatively recent standard of MPEG [1]. RVC itself does not define a new video compression methodology, but provides a unified way to describe already existing video codecs.

A unified codec specification approach requires one Model of Computation (MoC) and language, which in the case of RVC is CAL [2]. The CAL language offers a dataflow-based MoC, where individual nodes (called actors) communicate with each other through FIFO buffers. The set of nodes describing an application is a CAL network.

After having been selected as the official language of the RVC standard, a variety of development tools have been designed for the CAL language. The most notable of these is the Orcc compiler[1], which reads applications described in CAL, and generates VHDL, C, C++ or LLVM code as output.

Applications described in CAL are easy to parallelize and port to other platforms due to the generality and high level of expression of the language. However, there are also drawbacks in the description generality: implementation code (*e.g.* C) generated from CAL specifications is less optimized than hand-written implementation code. To alleviate the performance gap between hand-written implementations and those generated from high-level CAL specifications, a variety of approaches [3, 4, 5] have been proposed.

Here, we concentrate on the approach of quasi-static scheduling. It has been shown that quasi-static scheduling can produce considerable performance improvement [5], but unfortunately no fully automatic method has yet been proposed to create quasi-static schedules for CAL actor networks.

This paper presents an automatic approach for creating quasi-statically scheduled RVC implementations. The solution is partially based on the Orcc compiler and includes a code generator that automatically generates ready-to-compile quasi-static C code. In the experiments we present the clear performance benefit offered by the proposed approach.

2. STATEMENT OF THE PROBLEM

An RVC decoder is specified with a network of CAL actors. Each CAL actor represents a logical part of the video decoder (*e.g.* dequantizer, IDCT *etc.*). Internally, actors work like finite state machines (FSMs) that contain states, state transitions and internal variables. To enable conditional execution, state transitions of CAL actors may also have *guard conditions* that must be fulfilled for the state transition to happen.

Information is transferred from one actor to another over point-to-point FIFO buffers. The data is wrapped inside *tokens* and each FIFO carries tokens of a specified size. Connections between FIFOs and actors are called actor *ports*. Computations are performed in actor FSM state transitions called *actions* that produce and consume different numbers of tokens. When a state transition in an actor takes place, it is said that the actor *fires*. Fig. 1 depicts two CAL actors.

The CAL MoC is very powerful, but unfortunately enables functionality that makes the actor network hard to analyze. In contrast, for example, the synchronous data flow (SDF) MoC is straightforward and can be thoroughly analyzed [6]. The main difference between the CAL MoC and that of SDF is that SDF does not allow different paths of execution; every data item is processed by the same computa-

[1] Available from http://orcc.sf.net

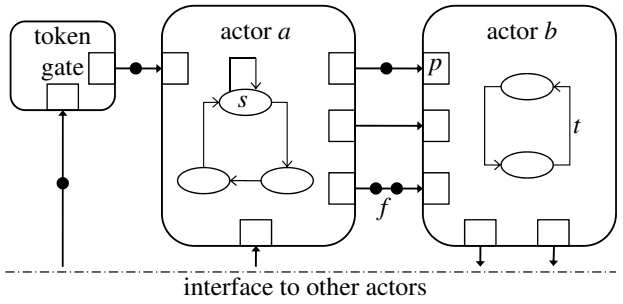

Fig. 1. Actors a and b, port p, FSM state s and state transition t. FIFO buffer f carries two tokens.

tions. In contrast, by the use of the aforementioned guard conditions, CAL allows the behaviour of actors to change dynamically. One further way to express this is by saying that SDF only allows *data tokens*, whereas CAL uses both data and *control tokens*.

This flexibility and unpredictability of CAL makes the language hard to analyze and complicates the computation scheduling greatly. By default, the Orcc compiler that produces C code from CAL specifications, creates a round-robin scheduler that iterates through all actors in the network and checks if there are actions that could be fired. This approach is fool-proof in its generality, but requires the scheduling algorithm to repeadetly poll for token availability, values of variables and states of actors.

Large CAL networks (such as video decoders) contain around one hundred actors and up to a few hundred FIFO buffers between them. When the CAL network is compiled, and executed on a processor, the FIFOs and variables necessarily reside in the main memory of the processor, to which random accesses are very slow. Thus, the superfluous polling of a round-robin scheduler adds a considerable amount of memory accesses and therefore increases the execution time.

In contrast, a quasi-statically scheduled CAL network is analyzed at compile time, and excess polling is removed prior to code generation. Thus, at run-time, the quasi-statically scheduled network runs in a more predictable fashion, with considerably less polling and thus, faster. The challenge of quasi-static scheduling is to find the parts in the CAL network that behave in a fully predictable manner.

3. PROPOSED APPROACH

In a previous publication [5], quasi-static scheduling was introduced in a more detailed fashion than above, but no automatic way was presented to acquire it. The fully automated approach presented in this paper can be divided into three independent parts: 1) analysis preparation, 2) CAL network analysis and 3) code generation.

The network analysis is performed by running the CAL network with real data and by recording the behaviour of each actor and the whole network to a kind of an execution trace. Based on the actor (and network) behaviour detected during this analysis, the code generator produces a quasi-static schedule to be used in the final run-time implementation.

3.1. Prerequisites

Enabling the run-time inspection of data values in a CAL network requires the insertion of *hooks* to actors. Evidently, this monitoring must not affect the network functionality.

An actor network having quasi-static behaviour means that there are data and control tokens flowing in the FIFO buffers. However, the CAL language does not contain different token types, which means that tokens affecting the network behaviour (control tokens) must be identified by special methods. The crucial question is: which FIFO buffers in the network carry control tokens?

Fig. 1 shows a toy example: FIFO f connects actors a and b. In the network analysis, we look at the structure of actor b and decide that FIFO f carries control tokens, if *values of tokens* coming from f can affect the action firing of actor b.

The presence or absence of tokens in f affects the possibility of firing actions as well, but we are not interested in this. Besides tokens, also internal variables of actors may affect actor firings. Thus, all variables that affect state transitions, are under analysis as well.

In practice, the analysis tool identifies all actions in each actor, that have conditional execution (guard conditions). These actions are monitored all the time when the network analysis is running.

3.2. Token gating

Essentially, a quasi-statically scheduled actor network works in different operating modes that are defined by the values of control tokens. It can be stated that the control tokens parameterize the network behaviour as in parameterized SDF [7].

This assumption implies that the control tokens truly have total control over the actor network. By default, this is not the case in a round-robin scheduled network. Thus, the network must be harnessed to be under control of the control tokens both in the analysis phase and the final quasi-static implementation. In practice we do this by adding *token gates* to FIFOs that control the network behaviour.

A token gate actor performs no computations, but simply blocks tokens from flowing when the gate is closed. The uses of the token gate are twofold: first, the gate orchestrates the computations so that a new computation starts only after the previous one has finished, and second, the value of the token passing through the gate can be observed and used as a parameter to define the behaviour of the network.

The feasible places for the token gate are those FIFO buffers that are observed to carry control tokens, as described in Subsection 3.1. In the set of CAL networks we used in the

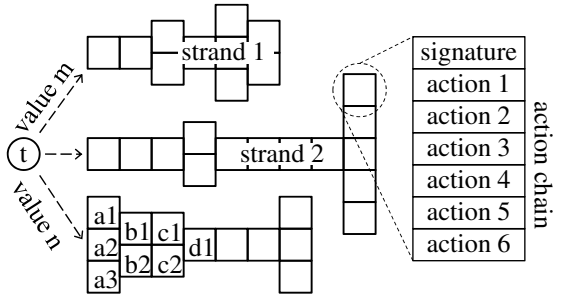

Fig. 2. Gate token *t* can have three different values, each invoking a specific strand. In strand 3, actor *a* has three possible action chains: *a1*, *a2* and *a3*.

experiments, the number of such FIFOs ranges between 5 and 30 in each network. Thus, all options can be iterated through to see which FIFO produces the best results. At present, our code generator allows only one token gate to be placed in each network.

In the end of the analysis preparation stage, the token gate CAL actor is inserted to the network after which it appears to the analysis tool just like any other actor. The token gate is also depicted in Fig. 1.

3.3. Network analysis

The dynamic analysis of the CAL network is performed at two different levels of hierarchy: the network level and the actor level. When the analysis is running, the relevant state parameters (we call *signature*) of each actor are recorded before the actor is allowed to operate. The actor signature contains: 1) the FSM state, 2) the number of tokens on each input port, 3) the value of the first token on each input port, 4) the value of each relevant internal variable. The internal variables are selected automatically as described in Subsection 3.1.

When the signature has been recorded, the actor is allowed to execute. Meanwhile, the analysis software records all the actions that the actor performs and finally observes the end FSM state of the actor. This recorded sequence of actions we call an *action chain*. Thus, there is one action chain for each signature.

Whilst running the analysis, each actor is invoked thousands of times. Between invocations, actors produce a number of different signatures and action chains. In some rare cases, it happens that for the same signature, an actor exhibits two different action chains. This means that the behaviour of the actor can not be modeled by our quasi-static assumption. In this case, that signature is labeled *dynamic* and triggers the basic scheduler at run-time in the final implementation.

The control tokens that flow through the token gate are called *gate tokens*. On the network level, the analysis software maintains a data record for each gate token value. The data record contains the sequence of actors that is invoked for that gate token value, and this sequence of actors is called a *strand*. Generally, actors in a strand may behave in various ways, so each actor slot in a strand has several alternative action chains that are identified based on their signature (see Fig. 2).

Upon completion, the analysis software has the following records: 1) a set of signatures for each actor, and an action chain that follows each signature; 2) one strand for each gate token value.

In practice, it required modifications to the Orcc code generator to allow the observation and recording of this data. Because the modified Orcc code generator still produces general-purpose C code, the running of this analysis takes only up to a couple of minutes despite the large amount of data it records.

3.4. Code generation

The code generator starts producing the quasi-statically scheduled code by iterating through every gate token value. For each gate token value, a C function is produced that contains the strand of actor invocations. As stated before, there are generally several different actor behaviours for each actor slot in a strand. This variation is taken care of at actor level.

For each actor slot in a strand, the code generator browses its databases to locate all possible actor behaviours that have taken place in this slot and gathers these as a set of signatures S. Then, the code generator analyzes the signatures in S and produces a set of rules to identify which action chain to fire. The rule may be, for example

```
if (port_1_token_count() == 64 &&
    variable_3_count() == 0 &&
    port_1_value() == 1024) then ...
```

Naturally, the code generator tries to use as few conditions as possible, as it is wasteful at run-time to unnecessarily check the conditions of variables and FIFO states. If an actor requires too many conditional statements to evaluate to a single action chain, or there are too many different actor behaviours, the code generator inserts a call to the default actor scheduler. In these cases, it would have taken more clock cycles to select the correct action chain than running the default scheduler does.

It is important to notice that these rules to select the correct actor behaviour are dependent on the gate token value and the actor index in the strand; *i.e.* if one actor is invoked twice in the same strand, both instances of the same actor have different sets of rules.

3.5. The runtime system

The quasi-statically scheduled CAL network is composed of C code that originates from two different sources. First, the schedule files that originate from our code generator, and second, the actor C files that are produced by another modified

Table 1. Speedup (s.up) provided by quasi-static scheduling (qss) compared to the default scheduling of the network. Numbers are gigacycles used to decode 500 frames.

Netw.	*s1*	*s2*	*m1*	*m2*	*l*	*s.up*
MVG	98.5	100.7	94.5	102.7	109.2	
qss	44.6	48.8	42.0	52.0	52.9	2.11
RVC	80.0	82.9	76.6	85.0	88.6	
qss	67.6	74.2	64.4	77.8	79.0	1.14
Serial	44.6	47.7	42.0	50.4	51.5	
qss	32.4	36.4	29.9	39.7	40.0	1.33
Xilinx	119.7	145.8	110.6	161.5	154.1	
qss	96.6	124.3	95.9	142.7	116.2	1.20

backend of the Orcc code generator. The special Orcc backend to produce the run-time network implementation provides access to the actor variables and FIFO contents that enable choosing between strands and action chains.

4. EXPERIMENTS

Our tool for automatic generation of quasi-static schedules was experimented on 4 different CAL networks. These networks were different implementations of the MPEG-4 Simple Profile video decoder.

The quasi-static schedules were acquired by using a 176x144 video sequence of up to 250 frames as training data. The decoding performance was measured with 5 different video sequences that had the resolution of 720x480 and lasted 500 frames. The experiments were performed on an Intel Core 2 Duo E8500 -based workstation running Windows 7. The compiler used was MS Visual Studio 2008 with default O2 level optimizations enabled.

The results are visible in Table 1. An example of reading Table 1 goes as follows: with the default Orcc scheduler and the MVG network, it takes 98.5 gigacycles (Gc) to decode 500 frames of the *s1* video sequence. With the quasi-static schedule generated by the tools described here, the same effort takes 44.6 Gc, providing a speedup of 2.21. Network-wise speedups are shown on the right and the overall average speedup is 1.45. The speedup differences between networks are rather great. The speedup that quasi-static scheduling can offer depends on the actors that are present in the network, as well as the network structure.

The tools described in this paper are available[2] as open source, and the networks are available from the Orcc website.

5. DISCUSSION

Although the experimental results already show a considerable performance improvement, it is evident that there is much more potential in the presented approach, than what has been shown.

The presented analysis approach is based on execution traces, which means that the outcome of the analysis depends on the input data. If the training data sequence does not utilize all allowed CAL network behaviours, the quasi-statically scheduled decoder will fail if such a previously unknown behaviour is invoked. A more robust approach would be to use static code analysis for the network. Static CAL code analysis performed by Wipliez and Raulet [3] could replace some preprocessing and analysis steps that have been described in this paper. This is a clear direction for future work.

6. CONCLUSION

In this paper we have described an automatic way of acquiring quasi-static schedules for CAL actor networks. The approach is based on dynamic code analysis followed by the generation of improved program code. Experiments show the quasi-statically scheduled code to provide an average program speedup of 1.5x.

7. REFERENCES

[1] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding standard," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 159–167, May 2010.

[2] J. Eker and J. Janneck, "CAL language report," Tech. Rep. UCB/ERL M03/48, UC Berkeley, 2003.

[3] M. Wipliez and M. Raulet, "Classification and transformation of dynamic dataflow programs," in *Conference on Design and Architectures for Signal and Image Processing*, 2010.

[4] R. Gu, J. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, pp. 1–14, 2010.

[5] J. Boutellier, C. Lucarz, S. Lafond, V. Martin Gomez, and M. Mattavelli, "Quasi-static scheduling of CAL actor networks for Reconfigurable Video Coding," *Journal of Signal Processing Systems*, 2009, DOI:10.1007/s11265-009-0389-5.

[6] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, New York, NY, 2000.

[7] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, Oct 2001.

[2] https://sourceforge.net/projects/efsmsched/